An overview of POSIX threads

Basics of POSIX Threads
programming on Linux

Jerry Feldman <gaf@blu.org>

What We will talk about

- What is a Thread.
- Why use threads. Thread vs. fork().
- Deadlock and Race Conditions
- Kernel threads vs. user threads. This is important in Linux 2.6+ kernels with NPTL.
- Basic thread concepts and functions: thread creation, thread join, mutex, conditions, attributes.
  - I won't go into details of attributes.
- Some examples. I have trimmed some of the examples to fit into the presentation
- References
- I'll post the presentation and examples on the BLU web site by early next week.

What is a thread

- In general, a thread is a separate context of execution that runs within a single program.
- Then, what is a process?
  - A process is a separate context of execution with respect to the host operating system. When you run a program, it runs as a separate process.

Then what is the difference?

- A process has its own memory and other attributes that make it independent.
- A thread is a part of an existing program. It shares its memory with the rest of the program and runs within the priority of the program.

Why do we have both

- A thread may be termed as a “light weight process”. Since the OS does not need to allocate memory and the other things it does with a process, starting a thread is cheaper, and faster. A thread is like a sports car in comparison to a process which is a truck.

Performance of Threads over fork.

- As I mentioned earlier, thread creation is cheaper than creating a process.

Package	Real Time	User Time	System Time
Fork()	9.555	0.938	8.521
Linux Threads	1.678	1.003	0.535
NPTL	1.648	0.984	0.574

Why use threads at all

- As I mentioned earlier, thread management takes up fewer system resources than does process management.
- A threaded application that abides by the POSIX standard is more portable between Linux and Unix as well as to other operating systems.
- Many things we do in programming can be done better concurrently. Note that almost all web browsers use threads.

What thread packages are available

- A number of different thread implementations have been used on Linux, but the most common is a POSIX implementation called LinuxThreads.
- With the arrival of the 2.6 kernel we also have the Native POSIX Thread Library (NPTL). This model performs better than the old LinuxThreads model and is more POSIX compliant.
- Most commercial Unix systems have either a POSIX compliant thread package, or a proprietary package, like Sun's LWP.

What is a Deadlock

- A deadlock is when two or more threads in your program are blocked from gaining access to a resource. The following example is a modification of Djikstra's Dining Philosophers.
  - John and David are at a table each with a bowl of spaghetti. One must have both a fork and a spoon to eat, but there is only one fork and one spoon. They must only take a single utensil at a time.
  - John takes the fork and David takes the spoon. Now, neither of them can eat unless we change the rules or they cheat.

How do we prevent deadlocks

- There are several ways to prevent deadlocks. One way is to define the order by which a resource is acquired. In our little John and David example, let's add a rule that one must take the fork first.
  1. John takes the fork before David can get it.
  2. John can then take the spoon.
  3. John eats
  4. John puts down the utensils, first the spoon, then the fork.
  5. David takes the fork
  6. Then he takes the spoon
  7. Then he gets angry and starts a food fight.
  8. He then puts down the utensils.
- The above method is one of the more common methods.

Now we have a Race Condition

- A race condition occurs when multiple threads access and manipulate the same data concurrently, and the outcome of the execution depends on the particular order in which the access takes place.
  - Take 2 threads john and David), each want to increment the value of a number which starts as 1.
    1. John and David both load the value, 1 at the same time.
    2. John and David increment the value.
    3. John and David store the value simultaneously.
    4. The result is 2!!!, but it should be 3.

How do we solve a Race Condition

- To solve the race condition we introduce a lock. In POSIX threads, this is called a MUTEX, or a Mutual Exclusion Primitive.
  1. John and David both try to lock the number, but John acquires the lock first. David waits.
  2. John loads the number.
  3. John increments it to 2.
  4. John stores it.
  5. John unlocks it.
  6. David now loads it.
  7. David then increments it to 3.
  8. David stores it.
  9. David releases the lock.
  10. The number is now 3.

Kernel vs. User Threads

- There are 3 basic threading models:
  1. User threads. Each thread is managed and dispatched in user space. The kernel does not get involved.
  2. Kernel Threads. In this case, the kernel manages each of the threads. Both LinuxThreads and NPTL use this model (called 1-on-1).
  3. Combination. The thread package uses the benefits of each. Most commercial Unix thread packages use this model (called M-on-N). This requires a great deal of additional support in the kernel.

Thread Safeness

- Thread-safe means that a function may be called concurrently by many threads without destructive results.
  - Some standard C library calls are not threadsafe because they may contain a static variable. For instance, the ctime(3) function returns a buffer containing the formatted time.

Thread safe example

- The following function is not thread safe:

```
char *notsafe()
{
        static char foo[128];
        /* populate foo */
        return foo;
}
```

- The following modification makes it threadsafe.

```
char *Threadsafe(char *foo, size_t foosize)
{
        /* populate foo */
        return foo;
}
```

The reason why the second function is threadsafe is that the calling thread passes the buffer into the function.

Errno

- In C, the variable, `errno`, is a global variable set by library functions and system calls. This is a bad thing for threads since it makes those functions who set it not thread safe.
- In threads, each thread gets its own errno thus preventing library functions from side affecting other threads.

The Thread API

- Before I present the basic thread API, I think it is important to define a few more concepts:
  - A thread object is the running thread.
  - A mutex is a mutual exclusion primitive, similar in concept to a semaphore.
  - An attribute is attached to either a thread or mutex, and does things like making the thread either joinable or detached. Or a mutex recursive.
  - A condition is an object where a thread waits until it receives a signal or a timeout.

General Thread API information

- Most functions return 0 on success and an error code on unsuccessful return. These functions do not set errno.

General Thread API information

- Most functions return 0 on success and an error code on unsuccessful return. These functions do not set errno.

Routine Prefix	Functional Group
pthread_	Threads themselves and miscellaneous subroutines
pthread_attr_	Thread attributes objects
pthread_mutex_	Mutexes
pthread_mutexattr_	Mutex attributes objects
pthread_cond_	Conditions
pthread_condattr_	Condition attributes objects

Creating a thread

- int pthread_create(pthread_t * thread,
  pthread_attr_t * attr,
  void *(*start_routine)(void *),
  void * arg);
  - Thread is a pointer to the thread object to be created.
  - attr is the thread attribute (or NULL for the default attribute).
    Most of the time, you will not set an attribute. One attribute you might set is ***`detachstate`***. The default for a thread is ***`joinable`***. A ***`joinable`*** thread remains in memory until joined. A ***`detached`*** thread resources are freed when the thread exits.
  - start_routine is the function where you want to start the thread. It's argument is a pointer, and it returns a pointer. The return value is only relevant for the *pthread_join*() function.
  - arg is a pointer to an argument. You can cast an integer and pass that directly if you want.

Terminating a thread

- You can terminate a thread in a few ways.
  - Thread returns from its ***start_routine***.
  - The thread calls ***pthread_exit().***
  - The thread is cancelled by a ***pthread_cancel***() call.
  - The entire process is terminated.
- void pthread_exit(void *retval);
  - Exits the thread and returns a value to its creator.

Joining a thread

- The only way to get the system to release the resources acquired by creating a joinable thread is to join them.
- int pthread_join(pthread_t th,
  void **thread_return);
  - The calling thread is blocked while waiting for thread, ***th,*** to complete.
  - If thread_return is not NULL, the return value of ***th*** is stored in the location pointed to by thread_return. The return value of ***th*** is either the argument it gave to pthread_exit(3), or PTHREAD_CANCELED if ***th*** was cancelled. It expects that the return value is a ***(void *).*** See the following example.

Simple Thread Example 1 of 2

```
#include <pthread.h>
#include <stdio.h>
#include <string.h>
#define NUM_THREADS     5

void *PrintThread(void *threadnum)
{
  printf("Thread number %d\n", (int)threadnum);
  pthread_exit(threadnum); /* return thread number */
}
```

Simple Thread Example 2 of 2

```
int main (int argc, char *argv[])
{
  pthread_t threads[NUM_THREADS]; /* for each thread */
  int rc, t; void *tv;
  for(t=0;t < NUM_THREADS;t++){
    printf("Creating thread %d\n", t + 1);
    rc = pthread_create(&threads[t], NULL, PrintThread, (void *)(t + 1));
    if (rc) {  fprintf(stderr, "pthread_create() is %s\n", strerror(rc));
      exit(-1);
    }
  }
  /* Threads are joinable release the resources. */
     for(t=0;t < NUM_THREADS;t++){
    rc = pthread_join(threads[t], NULL);
    if (rc){ fprintf(stderr, "pthread_exit() is %s\n", strerror(rc)); exit(-1); }
    else printf("Thread %d joined and returned %d\n", t+1, (int)tv);
  }
  return 0;
}
```

Synchronization

- As we saw in the race condition definition, we need to have some kind of locking device.
  - Threads calls them MUTEX (Mutual Exclusion)
    - Only one thread may own (or lock) a mutex at a time.
    - A fast mutex is one that can only be locked once by the owner.
    - A recursive mutex may be locked multiple times by the owner.
    - A error checking mutex causes a lock to return if it would block.

Mutex Creation and Initialization

- There are two ways to create and initialize a mutex
  1. Statically by using the built-in initialization constants.
     - `pthread_mutex_t fastmutex = PTHREAD_MUTEX_INITIALIZER;`
  2. Dynamically by calling pthread_mutex_init.
     - `pthread_mutex_t amutex;`
       `rc = pthread_mutex_init(&amutex, NULL);`
       Note that the second parameter is an attribute. You can create an attribute that sets the mutex up as recursive or error checking. Fast is the default.
     - `int pthread_mutex_destroy(pthread_mutex_t *mutex);`
       This destroys the mutex.

Locking and unlocking

- The following functions are used to lock and unlock a mutex.

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

Attempt to lock a mutex. If the mutex is already locked, the calling thread blocks. If the mutex is a recursive mutex, and the calling thread is the owner (currently holds the lock) this does not block).

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Unlocks the mutex.

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

Allows a thread to see if a mutex is locked. If it is not locked then the calling thread acquires the mutex, if it is locked by another thread it returns EBUSY.

Mutex Example 1 of 5

```
#include <pthread.h>     /* pthread functions and data structures */
#include <stdio.h>       /* standard I/O routines                 */
#define NUM_EMPLOYEES 2                  /* size of each array.    */
/* global mutex for our program. assignment initializes it */
pthread_mutex_t a_mutex = PTHREAD_MUTEX_INITIALIZER;
struct employee {
    int number;
    int id;
    char first_name[20];
    char last_name[30];
    char department[30];
    int room_number;
};
/* global variable - our employees array, with 2 employees */
struct employee employees[] = {
    { 1, 12345678, "danny", "cohen", "Accounting", 101},
    { 2, 87654321, "moshe", "levy", "Programmers", 202}
};
/* global variable - employee of the day. */
struct employee employee_of_the_day;
```

Mutex Example 2 of 5

```
/* function to copy one employee struct into another */
void copy_employee(struct employee* from, struct employee* to)
{
    int rc;      /* contain mutex lock/unlock results */
    /* lock the mutex, to assure exclusive access to 'a' and 'b'. */
    rc = pthread_mutex_lock(&a_mutex);
    to->number = from->number;
    to->id = from->id;
    strcpy(to->first_name, from->first_name);
    strcpy(to->last_name, from->last_name);
    strcpy(to->department, from->department);
    to->room_number = from->room_number;
    /* unlock mutex */
    rc = pthread_mutex_unlock(&a_mutex);
}
/* function to be executed by the variable setting threads thread */
void*do_loop(void* data)
{
    int my_num = *((int*)data);   /* thread identifying number         */
    while (1) {
        /* set employee of the day to be the one with number 'my_num'. */
                copy_employee(&employees[my_num-1], &employee_of_the_day);
    }
}
```

Mutex Example 3 of 5

```
/* like any C program, program's execution begins in main */
int
main(int argc, char* argv[])
{
    int       i;              /* loop counter                          */
    int       thr_id1;        /* thread ID for the first new thread    */
    int       thr_id2;        /* thread ID for the second new thread   */
    pthread_t  p_thread1;     /* first thread's structure              */
    pthread_t  p_thread2;     /* second thread's structure             */
    int       num1      = 1;  /* thread 1 employee number             */
    int       num2      = 2;  /* thread 2 employee number             */
    struct employee eotd;     /* local copy of 'employee of the day'.  */
    struct employee* worker;   /* pointer to currently checked employee */

    /* initialize employee of the day to first 1. */
    copy_employee(&employees[0], &employee_of_the_day);

    /* create a new thread that will execute 'do_loop()' with '1'       */
    thr_id1 = pthread_create(&p_thread1, NULL, do_loop, (void*)&num1);
    /* create a second thread that will execute 'do_loop()' with '2'    */
    thr_id2 = pthread_create(&p_thread2, NULL, do_loop, (void*)&num2);
```

Mutex Example 4 of 5

```
/* run a loop that verifies integrity of 'employee of the day' many */
    /* many many times.....                                          */
    for (i=0; i<60000; i++) {
        /* save contents of 'employee of the day' to local 'worker'.    */
        copy_employee(&employee_of_the_day, &eotd);
                worker = &employees[eotd.number-1];

        /* compare employees */
                if (eotd.id != worker->id) {
                    printf("mismatching 'id' , %d != %d (loop '%d')\n",
                                eotd.id, worker->id, i);
                    exit(0);
                }
                if (strcmp(eotd.first_name, worker->first_name) != 0) {
                    printf("mismatching 'first_name' , %s != %s (loop '%d')\n",
                                eotd.first_name, worker->first_name, i);
                    exit(0);
                }
```

Mutex Example 5 of 5

```
            if (strcmp(eotd.last_name, worker->last_name) != 0) {
                printf("mismatching 'last_name' , %s != %s (loop '%d')\n",
                        eotd.last_name, worker->last_name, i);
                exit(0);
            }
            if (strcmp(eotd.department, worker->department) != 0) {
                printf("mismatching 'department' , %s != %s (loop '%d')\n",
                        eotd.department, worker->department, i);
                exit(0);
            }
            if (eotd.room_number != worker->room_number) {
                printf("mismatching 'room_number' , %d != %d (loop '%d')\n",
                        eotd.room_number, worker->room_number, i);
                exit(0);
            }
    }
    printf("Glory, employees contents was always consistent\n");
    return 0;
}
```

Conditions

- A condition is a synchronization device that allows threads to suspend execution and relinquish the processors until some predicate on shared data is satisfied.
  - In essence a thread acquires a mutex, then waits on a condition until it receives a signal or a timeout. The mutex is automatically released while the thread is waiting, and then reacquired before it returns from the condition.
  - Another thread may signal a condition by waking up one or all of the threads.

Condition APIs

- pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
  Create a static condition (similar to mutex).
- int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond_attr);
  Initializes a dynamically created condition.
- int pthread_cond_destroy(pthread_cond_t *cond);
  Destroy a dynamically created condition.
- int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
  Wait on a condition for a signal to occur.
- int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime);
  Wait on a condition for a signal or a timeout to occur.
- int pthread_cond_signal(pthread_cond_t *cond);
  Signals the condition to release ONE thread.
- int pthread_cond_broadcast(pthread_cond_t *cond);
  Signal the condition to release All threads that may be waiting.
- Note that pthread_cond_wait and pthread_cond_timedwait are cancellation points.

A client server queuing example.

- The following example takes several arguments, specifically, an input file, an output file, the number of servers and the number of clients.
- A server reads the input file and places each line on a queue, and wakes up a client.
- A client then grabs a line from the queue, and writes it to the output file.

Queue example 1 of 24

```
#include <pthread.h>
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/times.h>
#include <sys/wait.h>
#include <assert.h>
#include <ctype.h>
char *strdup(const char *);
#if defined (PTHREAD_THREADS_MAX)
#   define MAXTHREADS PTHREAD_THREADS_MAX
#else
#   define MAXTHREADS 100
#endif
#define WAIT_TIME  2 /* 2 seconds default */
```

Queue example 2 of 24

```
/****************** QUEUE Operations ***********************/
/****************** QUEUE Data Structures ******************/
typedef struct llist {
  struct llist *next;
  void *data;
} LLIST;

typedef struct lqueue {
  LLIST *head, *tail;
  pthread_mutex_t mut;
  pthread_cond_t cond;
  volatile int count;
  int wait_time; /* Number of seconds to wait on queue */
} LQUEUE;
```

Queue example 3 of 24

```
/* CreateQueue */
LQUEUE *CreateQueue(int wait_time)
{
  int rv;
  LQUEUE *new = malloc(sizeof(LQUEUE));
  if (new) {
    new->head = new->tail = NULL;
    rv = pthread_mutex_init(&new->mut, NULL);
    if (rv) {
      free(new);
      return NULL;
    }
    rv = pthread_cond_init(&new->cond, NULL);
    if (rv) {
      pthread_mutex_destroy(&new->mut);
      free(new);
      return NULL;
    }
    new->count = 0;
    new->wait_time = wait_time;
  }
  return new;
}
```

Queue example 4 of 24

```
/*
 * QueueDestroy
 */
/* Assumes that queue is empty */
void QueueDestroy(LQUEUE *queue)
{
  pthread_mutex_destroy(&queue->mut);
  pthread_cond_destroy(&queue->cond);
  free(queue);
}
```

Queue example 5 of 24

```
/* AddDatatoQueue Always signal when adding data */
int AddDatatoQueue(void *data, LQUEUE *queue)
{
  LLIST *newlist = malloc(sizeof(LLIST));
  if (newlist == NULL)
     return -1;
  newlist->next = NULL;
  newlist->data = data;
  pthread_mutex_lock(&queue->mut); /* Acquire the queue */
  if (queue->count == 0) {
    queue->head = queue->tail = newlist;
  } else {
    queue->tail->next = newlist;
    queue->tail = newlist;
  }
  queue->count++;
  /* Release the queue and notify clients */
  pthread_mutex_unlock(&queue->mut);
  pthread_cond_signal(&queue->cond);
  return 0;
}
```

Queue example 6 of 24

```
/* RemoveDataFromQueue
 * This function is used to simply remove data from queue
 * Called by a thread that has not already acquired the lock */
void *RemoveDataFromQueue(LQUEUE *queue)
{
  LLIST *this;
  void *data = NULL;
  pthread_mutex_lock(&queue->mut);
  if (queue->count > 0) {
    this = queue->head;
    if (--(queue->count) <= 0) {
      queue->head = queue->tail = NULL;
    } else {
      queue->head = this->next;
    }
    data = this->data;
    free(this);
  }
  pthread_mutex_unlock(&queue->mut);
  return data;
}
```

Queue example 7 of 24

```
/* RemoveDataFromQueueNolock
 * This function is used to simply remove data from queue
 * Called by a waiting thread that must lock the queue first */
void *RemoveDataFromQueueNolock(LQUEUE *queue)
{
  LLIST *this;
  void *data = NULL;
  if (queue->count > 0) {
    this = queue->head;
    if (--(queue->count) <= 0) {
      queue->head = queue->tail = NULL;
    } else {
      queue->head = this->next;
    }
    data = this->data;
    free(this);
  }
  return data;
}
/************** END OF QUEUE OPERATIONS *******************/
```

Queue example 8 of 24

```
/********** General data structures *********************/
typedef struct {
  int tnum; /* Number assigned by creator */
  pthread_t tid; /* My thread id assigned by system */
  int verbose;
  int rv;       /* value I return */
  LQUEUE *queue; /* My queue */
  /* for servers, name of input file, for clients, pointer to
   * OFTYPE.
   */
  void *file;
} TARG;
typedef struct {
  char *filename;
  pthread_mutex_t mut;
} OFTYPE;
OFTYPE ofile = {
  NULL, PTHREAD_MUTEX_INITIALIZER
};
/********************** Global Data ***************************/
/* RunFlag:When zero, tells threads to quit.
 * Does not need to be protected by mutex */
static volatile RunFlag = 1;
TARG threads[MAXTHREADS];
```

Queue example 9 of 24

```
int IsWS(char *s)
{
  /* Bypass leading spaces */
  while(*s && isspace(*s)) ++s;
  return (*s == '\0');
}
/* This can side affect ibuf */
char *tdup(char *ibuf)
{
  char *sp;
  /* move to first non white */
  while(*ibuf && isspace(*ibuf))
    ++ibuf;
  if (*ibuf == '\0')
    return NULL;
  sp = ibuf + strlen(ibuf) -1; /* point to last char of string */
  /* decrement sp until it points to last non-white */
  while(isspace(*sp)) --sp;
  sp[1] = '\0';
  return strdup(ibuf);
}
```

Queue example 10 of 24

```
/* Server:
 * Read input file until end of file.
 * place each record on the queue, let a */
static void *Server(void *argp)
{
  TARG *targ = argp;
  FILE *ifile;
  struct timespec abstime;
  char buf[256], *data;
  int rv, added = 0;
  if (targ->file == NULL) {
    fprintf(stderr, "S%03X:No file name\n", targ->tnum);
    targ->rv = -1;
    pthread_exit(targ);
  }
  if (targ->queue == NULL) {
    fprintf(stderr, "S%03X:No queue\n", targ->tnum);
    targ->rv = -1;
    pthread_exit(targ);
  }
  if (targ->verbose) fprintf(stderr, "S%03X:running\n", targ->tnum);
```

Queue example 11 of 24

```
ifile = fopen(targ->file, "r");
 while(fgets(buf, sizeof(buf), ifile)) {
   if (IsWS(buf))
     continue;
   data = tdup(buf); /* make a trimmed copy */
   if (data == NULL) {
     fprintf(stderr, "S%03X:Unable to allocate:%s\n",
                     targ->tnum,
                     strerror(errno));
     targ->rv = -1;
     pthread_exit(targ);
   }
   added++;
   AddDatatoQueue(data, targ->queue);
   rv = pthread_cond_signal(&targ->queue->cond);
   if (rv) {
     fprintf(stderr, "S%03X:Error on cond signal:%s\n",
                   targ->tnum, strerror(rv));
   }
 }
 fclose(ifile);
```

Queue example 12 of 24

```
if (rv) {
   fprintf(stderr, "S%03X:Error on queue mutex:%s\n",
                   targ->tnum, strerror(rv));
   free(data);
   targ->rv = -1;
   return targ;
 }
 while(RunFlag && targ->queue->count > 0) {
   gettimeofday((struct timeval*)&abstime, NULL); /* Recompute wait time */
   abstime.tv_sec += targ->queue->wait_time; /* Time out after 10 seconds */
   abstime.tv_nsec = 0;
   rv = pthread_cond_timedwait(&targ->queue->cond,
              &targ->queue->mut,
              &abstime);
   if (rv && rv != ETIMEDOUT) {
     fprintf(stderr, "S%03X:Error on cond wait:%s\n",
                   targ->tnum, strerror(rv));
     break;
   }
 }
```

Queue example 13 of 24

```
pthread_mutex_unlock(&targ->queue->mut);
  RunFlag = 0;
  pthread_cond_broadcast(&targ->queue->cond);
  if (targ->verbose)
    fprintf(stderr, "S%03X:Normal Exit, %d records queued\n",
                    targ->tnum, added);
  targ->rv = 0;
  pthread_exit(targ);
}
```

Queue example 14 of 24

```
/* writes data to file, frees data */
static int process_data(int tnum, OFTYPE *ofile, void *data)
{
  FILE *fp;
  int rv;
  fp = fopen(ofile->filename, "a+");
  if (fp == NULL) {
    fprintf(stderr, "C%03X:Error on open %s:%s\n",
                    tnum, ofile->filename, strerror(errno));
    free(data);
    return -1;
  }
  rv = pthread_mutex_lock(&ofile->mut);
  if (rv) {
    fprintf(stderr, "C%03X:Error on file mutex:%s\n",
                    tnum, strerror(rv));
    free(data);
    return -1;
  }
  fprintf(fp, "C%03X:%s\n", tnum, data);
  fclose(fp);
  pthread_mutex_unlock(&ofile->mut);
  free(data);
  return 0;
}
```

Queue example 15 of 24

```
/*
 * Wait on queue until either signalled or done.
 */
static void *Client(void *argp)
{
  int rv;
  pthread_t tid;
  TARG *targ;
  struct timespec abstime;
  LQUEUE *queue;
  void *data;
  int processed = 0;
  int timedout = 0;
  targ = (TARG *)argp;
  targ->rv = 0;
  queue = targ->queue;
  if (targ->verbose)
    fprintf(stderr, "C%03X:running acquiring\n", targ->tnum);
  rv = pthread_mutex_lock(&queue->mut);
  if (rv) {
    fprintf(stderr, "C%03X:Error on mutex:%s\n",
                    targ->tnum, strerror(rv));
    targ->rv = -1;
    pthread_exit(targ);
  }
```

Queue example 16 of 24

```
if (targ->verbose)
  fprintf(stderr, "C%03X:running acquired\n", targ->tnum);
 /* We loop on the predicate. cond_timedwait can wakeup
  * for other reasons
  */
 while(RunFlag) {
   /* Remove elements from queue */
   while (targ->queue->count > 0) {
     pthread_cond_broadcast(&queue->cond);
     if ((data = RemoveDataFromQueueNolock(queue))) {
            pthread_mutex_unlock(&queue->mut);
            /* allow another thread to run */
            pthread_cond_signal(&queue->cond);
            ++processed;
            process_data(targ->tnum, targ->file, data);
            /* Reacquire lock for the cond wait */
            pthread_mutex_lock(&queue->mut);
     }
   }
```

Queue example 17 of 24

```
/* Recompute wait time */
   gettimeofday((struct timeval*)&abstime, NULL);
   abstime.tv_sec += targ->queue->wait_time; /* Time out after n seconds */
   abstime.tv_nsec = 0;
   /* Recheck predicate before waiting */
   if (RunFlag == 0) {
     break;
   }
   rv = pthread_cond_timedwait(&queue->cond,
                                                &queue->mut,
                                                &abstime);
   if (rv && rv != ETIMEDOUT) {
     fprintf(stderr, "C%03X:Error on cond wait:%s\n",
                     targ->tnum, strerror(rv));
     targ->rv = 0;
     break;
   } else if (rv == ETIMEDOUT) {
     rv = 0;
     timedout++;
   }
 }
```

Queue example 18 of 24

```
pthread_mutex_unlock(&queue->mut);
 if (targ->verbose) {
   if (rv == 0) {
     if (timedout)
             fprintf(stderr, "C%03X:Successful completion, %d records, %d timeouts\n",
                     targ->tnum, processed, timedout);
     else
             fprintf(stderr, "C%03X:Successful completion, %d records\n",
                     targ->tnum, processed);
   }
 }
 pthread_exit(targ);
}
```

Queue example 19 of 24

```
static char *iam;
void usage(const char *iam)
{
  fprintf(stderr, "Usage: %s -s n -c n [ -v ] [ -w n ] input-file output-file where\n", iam);
  fprintf(stderr, "\t-s\tNumber of servers (must be > 0)\n");
  fprintf(stderr, "\t-c\tNumber of clients (must be > 0)\n");
  fprintf(stderr, "\t-w\tWait time for client queue\n");
  fprintf(stderr, "\t-v\tRun verbosely\n");
}
```

Queue example 20 of 24

```
int main(int argc, char **argv)
{
  int i;
  int fd, rv, c;
  int clients = 0;
  int servers = 0;
  char *ifile = NULL;
  int wait_time = WAIT_TIME;
  void *trv;
  struct timespec abstime;
  /* nthreads == number of child processes */
  int nthreads = 0;
  LQUEUE *queue;
  pthread_t tid;
  int thread_num = 0;
  int rq;
  int verbose = 0;
  if (iam = strrchr(argv[0], '/'))
    ++iam;
  else
    iam = argv[0];
  tid = pthread_self(); /* Get my threadid */
```

Queue example 21 of 24

```
while((c = getopt(argc, argv, "vc:s:w:")) != -1) {
  switch(c) {
  case 'v':
    verbose = 1; break;
  case 'c':
    clients = atoi(optarg); break;
  case 's':
    servers = atoi(optarg);  break;
  case 'w':
    wait_time = atoi(optarg);  break;
  default:
    usage(iam);
    return -1;
  }
}
if (optind < argc)
  ifile = argv[optind++];
if (optind < argc)
  ofile.filename = argv[optind++];
if (ifile == NULL || ofile.filename == NULL) {
  usage(iam);
}
```

Queue example 22 of 24

```
queue = CreateQueue(wait_time);
 assert(queue);
 nthreads = servers + clients;
 if (nthreads > MAXTHREADS ||
     servers <= 0 || clients <= 0) {
   usage(iam);
   return -1;
 }
 /* Start clients */
 for(thread_num=0;thread_num < clients;++thread_num) {
   threads[thread_num].tnum      = thread_num + 1;
   threads[thread_num].verbose   = verbose;
   threads[thread_num].queue     = queue;
   threads[thread_num].file      = &ofile;
   if(rv = pthread_create((pthread_t *)&threads[thread_num].tid,
                NULL, Client,   (void *)&threads[thread_num])) {
     fprintf(stderr, "Client thread %d create failed:%s\n",
                     thread_num+1, strerror(rv));
     /* Set last good thread for join */
     RunFlag = 0;
     pthread_cond_broadcast(&queue->cond);
     break;
   }
 }
```

Queue example 23 of 24

```
/* Start servers */
 for(i=0;i < servers;++i, ++thread_num) {
   threads[thread_num].tnum      = thread_num + 1;
   threads[thread_num].verbose   = verbose;
   threads[thread_num].queue     = queue;
   threads[thread_num].file      = ifile;
   if(rv = pthread_create((pthread_t *)&threads[thread_num].tid,
                                       NULL,
                                       Server,
                                       (void *)&threads[thread_num])) {
     fprintf(stderr, "Server thread %d create failed:%s\n",
                   thread_num+1, strerror(rv));
     /* Set last good thread for join */
     RunFlag = 0;
     pthread_cond_broadcast(&queue->cond);
     break;
   }
 }
```

Queue example 24 of 24

```
if (RunFlag == 0) {
   fprintf(stderr, "%d threads created, error, run is false\n", thread_num);
 } else if (verbose) {
   fprintf(stderr, "%d servers, %d clients running\n",
                    servers, clients, thread_num);
 }
 /* now wait for join */
 for(i=0;i<thread_num;i++) {
   rv = pthread_join(threads[i].tid, &trv);
   if (rv) {
     fprintf(stderr, "Thread %d join failed:%s\n",
                     threads[i].tid, strerror(rv));
   } else {
     rv = ((TARG *)trv)->rv;
     if (rv < 0) {
                fprintf(stderr, "Thread %d returned %d\n", threads[i].tnum, rv);
     } else if (verbose) {
                fprintf(stderr, "join success on %d\n", threads[i].tnum);
     }
   }
 }
 QueueDestroy(queue);
 return 0;
}
```

References
I used some material from these

- Programming with POSIX® Threads, Dave Butenhof, Addison-Wesley.
- POSIX Thread Programming
  http://www.llnl.gov/computing/tutorials/workshops/workshop/pthreads/MAIN.html
- Getting Started With POSIX Threads
  http://dis.cs.umass.edu/~wagner/threads_html/tutorial.html
- Programming POSIX Threads
  http://www.humanfactor.com/pthreads/
- YoLinux Tutorial: POSIX thread (pthread) libraries
  http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html
- Multi-Threaded Programming With POSIX Threads
  http://users.actcom.co.il/~choo/lupg/tutorials/multi-thread/multi-thread.html
- Time for Cambridge Brewery